COMSC-210 Lecture Topic 4
Using Dynamic Memory

□ Reference
Childs Ch. 4, 5

□ 3 Ways That Data Structures Store Values
"static" arrays
  e.g., `int score[100];`
"dynamic" arrays
  e.g., `int* score = new int[n];`
  e.g., `delete [] score;`
both static and dynamic are fixed-size
"linked lists" (topics 6 and 9)

□ Using Dynamic Arrays In Templates
"capacity" no longer in template specification:
  `template <class T>`
declaration: `DynamicArray<int> a;` // of default capacity
...or: `DynamicArray<int> a(10);` // of specified capacity
replace these data members from "StaticArray":
  `T values[CAPACITY];` // allocated right away
  `bool inUse[CAPACITY];` // set to false in constructor
with these data members in "DynamicArray":
  `static const int INIT_CAP = 100;` // initial capacity
  `int cap;` // avoid name conflict with "int capacity() const"
  `T* values;` // allocated in constructor
  `bool* inUse;` // allocated in constructor & set to false

□ The "Default" Constructor
prototype: `DynamicArray(int=INIT_CAP);`
  uses a "default parameter"
function (not inline):

```
template <class T>
DynamicArray<T>::DynamicArray(int init_cap)
{
  cap = init_cap;
  values = new T[cap];
  inUse = new bool[cap];

  for (int i = 0; i < cap; i++)
    inUse[i] = false;
}
```

□ Managing Dynamic Memory
include these functions:
1. Destructor
2. Copy Constructor
3. Assignment Operator

□ Templated Destructor
purpose: deallocate memory allocated in constructor
prototype: `~DynamicArray();`
function definition:

```
template <class T>
DynamicArray<T>::~DynamicArray()
{
  delete [] values;
  delete [] inUse;
}
```

□ Templated Copy Constructor
purpose: allows copies to have their own arrays
  avoids conflicting destructors
this happens when:
  pass-by-value: `void fun(DynamicArray<int>);`
  assignment: `DynamicArray<int> copy = a;`
  return value: `DynamicArray<int> fun();`
prototype: `DynamicArray(const DynamicArray<T>&);`
function definition:

```
template <class T>
DynamicArray<T>::DynamicArray(const DynamicArray<T>& a)
{
  cap = a.cap;
  values = new T[cap];
  inUse = new bool[cap];

  for (int i = 0; i < cap; i++)
  {
    values[i] = a.values[i];
    inUse[i] = a.inUse[i];
  }
}
```

□ Templated Assignment Operator
purpose: allows copies to have their own arrays
  avoids conflicting destructors
this happens when:
  assignment: `DynamicArray<int> copy; copy = a;`
prototype: `DynamicArray<T>& operator=(const DynamicArray<T>&);`
  returns a "self-reference"; allows `fun(copy = a);`
function definition:

```
template <class T>
DynamicArray<T>& DynamicArray<T>::operator=(const DynamicArray<T>& a)
{
  if (this == &a) return *this;

  // same as destructor code block
  ...

  // same as copy constructor code blocks
  ...

  return *this;
}
```

□ Overcoming Fixed Size
in StaticArray operator[ ] setter:
  if key >= capacity, return dummy
in DynamicArray operator[ ] setter:
  if key >= capacity, *increase capacity*
`key < 0` still returns the dummy
operator[ ] getter is unaffected

□ Increasing Capacity In `operator[]` setter
to accommodate `key >= cap`
reset `cap` to `key + 1` like this:

```
T* tempValues = new T[key + 1];
for (int i = 0; i < cap; i++) tempValues[i] = values[i];
delete [] values;
values = tempValues;

bool* tempInUse = new bool[key + 1];
for (int i = 0; i < cap; i++) tempInUse[i] = inUse[i];
for (int i = cap; i <= key; i++) tempInUse[i] = false;
```

```
            delete [] inUse;
            inUse = tempInUse;

            cap = key + 1;
```